Fall 2013 Midterm Solutions (Beta Edition)

0. So it begins. Circle the correct stuff and do all the things.

1. Union Find.

(a) P, I, I, I

(b) Path compression ensures that after an object's connectedness is tested, its it is set equal to its. After making a connected call involving every object, all objects point directly at their root and thus the depth of every node is at most 1. The tree height is thus $\Theta(1)$.

2. Analysis of Algorithms.

(a) Arrays.sort uses merge sort for objects. In the worst case, merge sort is $\Theta(N \log N)$.

(b) $\Theta(N \log N)$. First we need to know the runtime of the nested while loops. The code pattern here is very similar to the code pattern used in the collinear assignment for detecting runs in an array. We observe that after the inner while loop terminates, the control variable i is set equal to j, and j always starts just ahead of i. You can think of the loop as proceeding forward like an inch-worm: first the head moves forward a bit, and then the rear comes up to meet the head. Consequently, the runtime for the loops is simply linear.

Since the total run time is linear for the nested while loops, the total run time is dominated by the sorting step. The entire code fragment takes time, just like the inner loop for FastCollinearPoints.

(c) This can be done by first adding each string to a hash-based set of strings with initial size equal to the number of strings. One then simply iterates through the hash table and inserts items into a queue.

If you weren't sure how sets work, you could have also added the strings to a hash-based symbol table with dummy values.

3. Quicksort.

(a) FEED DIRT TO SWIMP
(but he won't like it)

(b) B: If all keys are equal, the pivot always ends up in the middle, so the run time is $\Theta(N \log N)$ in the worst case.

A: 3-way quicksort will perform only a single partitioning step, and that partition will take only linear time.

C. The pivot will always either end up at the front or back (depending on how you code up your bad quicksort). This is bad.

B. If all keys are unique, then it doesn't matter what we do with equal keys. Thus we're back to the usual best case for Quicksort.

4. **Heaps and Priority Queues.**

(a) `C D E E F F` **`K`** `Z W J X G H`

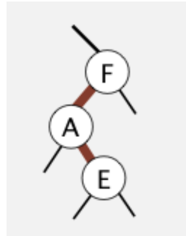
Due to a typo on the exam, it was also ok to have E in place of the K.

(b) $\Theta(1)$: In the best case, the new search node has higher priority than all other nodes. This can certainly happen if the next board is worse than all previous boards. In this case, the node fails to swim even a single spot, and the insert takes only constant time.

(c) $\Theta(S)$: Extra tricky problem. If we're using an array, and our API does not restrict the size, then on a real machine (like the one you used for 8puzzle), then you'll need to do some resizing. Partial credit for $\Theta(\log N)$.

5. **LLRBs.**

a. In the worst case, we perform 3 elementary operations. This was also referred to as case 3 in the lecture slides. It is also somewhat evident from the code for LLRBs. What we want is the structure given by:



b. 3 1 2 gives us this easily. 1 3 2 also gets us to case 3, but in a slightly more roundabout fashion since the process starts with a non-splitting related rotation after inserting 3.

c. B. The largest key is ALWAYS a right child, so must be black.

C. A two node 2-3 tree has a red smallest child. A three node 2-3 tree has a black smallest child.

B. We cannot have two consecutive red nodes.

C. A node with two children of the same color (which must be black) can be either red or black.

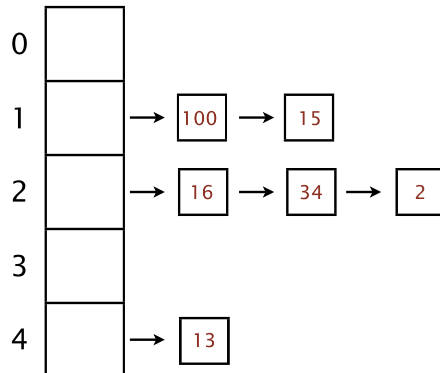
C. When the insertion operation for a BST has completed, the node node may be either red or black depending on rotations and color flips. Because of some ambiguity in the problem statement, we also accepted A.

6. **Hash Tables.**

a.

Key	Value	hashCode()	Index
"13"	A	4	4
"15"	B	6	1
"2"	C	2	2
"34"	D	7	2
"16"	E	7	2
"100"	F	1	1

b.



c. The new size is 10, so the hash function for our hash table becomes hashCode() % 10. Since every hashCode is less than 10 (for these keys), we have that each key ends up in the bin corresponding to exactly its hash code since the modulus operation does nothing.

Thus, after reinserting everything into the table, the only duplicates are "34" and "16", so the longest linked list is only size 2.

d. "1", "01", "10", and "100" all have the same hashCode() and thus will always be in the same bin, no matter the modulus.

e. We accepted either yes or no on the merits of your argument. No is a reasonable answer since the lookup time is going to be very quick if all linked lists are less than around size 5. Yes is a reasonable answer because performing log 5ish compares per search is better than 5ish compares per search, particularly if the code using the hash table is constrained by the lookup speed. We might also expect an improvement if probes are expensive, since our linked list nodes could be in very distant pieces of memory.

f. $\Theta(N^2)$. Our separate chaining implementation resizes by reinserting each item into a brand new symbol table. If all items hash to the same value, inserting N items into a linked list will take 1 + 2 + 3 + … + N-1 time, since each insert into a separate chaining hash table involves scanning each linked list to avoid duplicates.

7. **Sorting.**
   a. yes, yes, yes, no, no.
   b. A. If we want to sort a set of randomly ordered items such that we get the best performance and we don't care about stability, we should use quicksort.

      A or C. Again, we just want speed, but don't care about stability. If the Observations are randomly ordered, quicksort is the winner. It was also reasonable to assume that the unsorted Observation array was filled in roughly by timestamp, in which case we'd want to use insertion sort to take advantage of the partially ordered nature of the array.

      B. In this case, we want speed and stability, and our objects are randomly ordered with respect to importance. The winning sort here is mergesort.

      C. Here we have an array that is almost perfectly ordered, so we should use insertion sort.

8. **Extrinsic Max PQ.**

   Our data structures must support two basic operations: Locating and removing the maximum priority item (for delMax()), and looking up an existing item in order to set its priority (for put() to be able to update priorities). Both of these lookup operations must be completed in amortized log N time.

   There are two natural data structures for tracking the maximum priority item, i.e. implementing a priority queue: A heap and an LLRB

tree. Using either of these data structures ordered by priority will provide an easy and fast delMax() method, but will be too slow when updating the priorities of existing items, since we will have to search the entire structure to find the item whose priority needs updating.

This shortcoming suggests a natural approach where we have a symbol table that maps Items to their position in the priority-tracking data structure. Using either an LLRB based symbol table or a hashing based symbol table is fine. The latter is OK only if we assume uniform hashing and well-behaved data. An LLRB has better guaranteed performance.

Sample Solution 1: Using an array-based heap for tracking priorities.

Data structure description:

Array-based heap of nodes. A node is a helper class containing an int priority, an Item, and the index of the item in the heap.

Also use an LLRB based symbol table that maps Item to node.

put() description:

We look up the Item in the symbol table. If it does not exist, we add it to the symbol table, and create a new heap node containing the Item, priority, and index corresponding to the last entry of the heap. We then swim the item, adjusting the array indices of each node as we exchange.

If the Item does exist, we change the priority of the existing node, and sink the node if its priority has been decreased, and swim the node if it has been increased. Even though the symbol table only provides a link to the node (and not the array position), we can still perform the exchanges in the array because the node contains its array index.

delMax() description: Delete the max node from the priority queue, and bring the new max to the top as usual, correcting indices in each node for each sink operation completed. Delete the corresponding item from the symbol table.

Common mistake #1: Having nodes that contain only an item and priority, and a symbol table that maps to either node or index. The first choice (mapping to a node) is bad because if you only have a reference to the node, you don't know where it is in the array and cannot swap. The second choice (mapping to an array) is bad because

the symbol table references into the heap get messed up when you perform exchanges. There are many possible fixes, one of which is described above. An easier fix is to use a pointer based heap instead of an array based heap, at the expense of some memory and speed.

Common mistake #2: Mapping from Item to integer priority. This doesn't actually help since you still have to look it up in your heap, which is linear time.

Sample Solution 2: Using an LLRB for tracking priorities.

Data structure description:

LLRB based priority queue of nodes. A node is a helper class containing an int priority and an Item. Nodes are comparable on priority, and ties are broken based on Item.

LLRB based symbol table that maps Item to node.

put() description:

Look up the Item in the symbol table. If it exists, delete the corresponding node from the LLRB based priority queue. Insert a new node into the LLRB based priority queue with the same Item and new priority. Update the link in the symbol table to refer to this new node.

If the Item does not exist, create a new node and add it to the LLRB based priority queue, and create a mapping from the Item to this new node.

delMax() description:

Remove the max Item from the LLRB (can be completed in log N time by simply following all the right links, though you did not need to specify this detail). Look up the Item in the symbol table and delete it from the symbol table.

Common mistake: Not handling duplicate priorities. The LLRBs discussed in class do not allow duplicate keys. We avoided this problem in the sample solution by creating nodes that broke ties by Item.

More severe mistake:
Only storing priorities in the priority queue. When you delete the max, you don't know which Item to return.